Notes on Bit Fiddling in Java

Stephen M. Watt

October 24, 2009

This note explains how to work with bit fields in integer types in Java. This is useful to do low-level programming where several pieces of information are crammed into a single machine word.

Integer Representation

The first thing we need to do is understand how integers are represented.

Java has four sizes of integers:

- `byte` 8 bits (1 byte)
- `short` 16 bits (2 bytes)
- `int` 32 bits (4 bytes)
- `long` 64 bits (8 bytes)

Each of these stores signed integer values in a certain range. For a type that has *N* bits (e.g. `int` has *N*=32), the range of possible values that can be stored is $-2^{N-1} .. 2^{N-1} - 1$. So a value of type `byte` can take values in the range -2^7 to $2^7 - 1$, that is -128 to 127.

For an integer type with *N* bits we can number these bits 0 to *N*-1. For this note, we will use the subscript notation b<i> to mean bit i in the value b. We won't use square brackets just to emphasize that these are not arrays and we cannot really index into values to get at the bits.

How will we number the bits in a stored integer value? From the low addressed byte first? From the high addressed byte first? In what order within the bytes? For our purposes, it doesn't really matter. If all we want to do is to cram a bunch of small values together and be able to get at them later, the order the bits are stored does not matter.

Remember that any positive integer *K* less than 2^{N-1} can be written as $K = \sum_{i=0}^{N-2} K_i 2\hat{}i$, with each coefficient K_i equal to 0 or 1. We will adopt the convention that we number bits so that K<i> = K_i.

For example, if K = 21, we have K = 16 + 4 + 1 so K<0> = 1,K<1> = 0, K<2> = 1, K<3> = 0, K<4> = 1, and all other K<i> = 0.

For our purposes, we don't need to worry how negative numbers are stored.

Integer Constants

In Java integer constants can be entered in base 10 as usual, in base 8 by starting the number with a 0, or in base 16, by starting the number with 0x. For example 100 (which is 64 + 32 + 4) can also be written 0144 (i.e. $8^2 + 4 * 8^1 + 4 * 8^0$) or 0x64 (i.e. 6 * 16 + 4).

This is useful to create particular bit patterns. By writing numbers in base 16 you can create any pattern of zeros and ones easily, as follows:

Step 1. Write out the bit pattern you need as a base 2 number, e.g. 1010000011111100.

Step 2. Break the number into groups of 4 bits. E.g., for the number above 1010,0000,1111,1100.

Step3. Replace each group of 4 bits by its value as an integer, e.g. 10, 0, 15, 12.

Step 3. Write each of the numbers (now in the range 0..15) as a hexadecimal digit, and put 0x on the front, e.g. 0xA0FC.

Finally, to specify a 64 bit integer constant, you should put an "L" on the end of the number. For example, you can write 2000000000000000L.

Building Bit Patterns

We are going to need to be able to construct arbitrary bit patterns of our own choosing. To do this, we will need to use Java's shift operations << and >> as well as the bitwise logical operations &, | and ~.

H = K << m creates a new value by shifting the bits of K up by m positions. That is H<i+m> = K<i> for i = 0..N-1-m. Zeros are shifted in the bottom, so H<i> = 0 for i = 0..m-1. For example 0xF1A << 4 is 0xF1A0.

H = K >>> m creates a new value by shifting the bits of K down by m positions. That is H<i-m> = K<i> for i = m..N-1. Zeros are shifted in the top, so H<i> = 0 for i = N-m to N-1.

It is often useful to create a "mask" which has a group of 1 bits in a certain range, with the rest being zero. For example, suppose we wanted to make a mask M with M<12> = 1 and the other bits zero. Then we would create M = 1 << 12. If we wanted M<12>= M<13> = M<14> = M<15>= 1 and the other bits zero. Then we would need to shift up the number with four 1 bits by 12 positions. The number with four 1 bits is 0xF, so we would create M = 0xF << 12. If we needed a 64 bit mask we would write 0xFL << 12.

H = ~K creates the bitwise complement of K. That is H<i> = 0 if K<i> = 1 and H<i> = 1 if K<i> = 0.

H = K & G creates the bitwise "and" of K and G. That is H<i> = 1 if and only if K<i> = G<i> = 1.

H = K | G creates the bitwise "or" of K and G. That is H<i> = 1 if either of K<i> or G<i> = 1.

Jamming integers together

Suppose we wanted to save three smallish positive integers X, Y and Z into a long value, and that each of these integers was 20 bits or less. Then in a result R, we would put X into bits 0..19 of R, Y into bits 20 to 39 of R and Z into bits 40 to 59 of R. Bits 60 to 63 of R would not be used. We could form R as

```
R = X | (Y << 20) | (Z << 40).
```

Now suppose we wanted to get the values of X, Y and Z out of R. We would need a mask with twenty 1 bits. That is 0xFFFFF, using the method above. So we could get the values using

```
X = R & 0xFFFFF;
Y = (R & (0xFFFFFL << 20)) >>> 20;
Z = (R & (0xFFFFFL << 40)) >>> 40;
```

The L suffix was needed in the last two statements because the resulting values could not be represented in 32 bits.

More about Masks

In practice things might be complicated by the fact that the bit ranges of the mask might not end on boundaries that are multiples of four bits (and hence easily represented by F's). Just remember that you can leave off a few bits at the high end by using the hex digits 1, 3 (= 0011 base 2), 7 = 0111 base 2). So, for example, you could make a mask with 11 ones using 0x7FF and you could slide it up to positions 5 to 15 using 0x7FF << 5.

Changing a Bit Field

Suppose we have a number in which we are using bit fields and we want to change just one of the fields. The first thing we might think of would be to use |, the bitwise OR, for example

```
R = R | (Y << 20);   // WRONG!
```

As the comment says, this would be wrong. Why? Well,suppose some of the bits in R in the relevant range had the value 1. Then after the bitwise OR, they would still have the value one, *regardless of the value of the bits in Y.*

It is necessary to clear the bits in the applicable zone first. This can be done using bitwise AND with a mask that is 1 everywhere except the zone that is to be zeroed. Such a mask can be created using the bitwise complement. For example,

```
R = R & ~(0xFFFFFL << 20);   R = R | (Y << 20);
```

These statements could be written more concisely as

```
R &= ~(0xFFFFFL << 20);   R |= Y << 20;
```